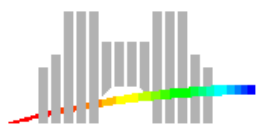
L'arithmétique des ordinateurs du néolithique à nos jours

Florent de Dinechin, projet Arénaire

RNAIM, janvier 2007, Montpellier



Avertissement

Le présent survol est centré autour du nombril de l'orateur.

Rien qu'en France, il y a beaucoup plus que présenté ici.

Je présente par avance mes excuses à mes estimés collègues.

Des petits cailloux aux ordinateurs : introduction historique

Des petits cailloux aux ordinateurs : introduction historique

Représentations modernes des entiers

Représentations des réels

Conclusion : l'arithmétique de mon PC

Néolithique

Invention “par les bergers du néolithique”

- du système de numération unaire

Néolithique

Invention “par les bergers du néolithique”

- du système de numération unaire
  - un caillou, un mouton
  - pas besoin de savoir compter pour manipuler des nombres

Néolithique

Invention “par les bergers du néolithique”

- du système de numération unaire
  - un caillou, un mouton
  - pas besoin de savoir compter pour manipuler des nombres
- de l’addition

Néolithique

Invention “par les bergers du néolithique”

- du système de numération unaire
  - un caillou, un mouton
  - pas besoin de savoir compter pour manipuler des nombres
- de l’addition
  - algorithme : verser un sac de caillou dans l’autre
  - rigolez pas, il y a un papier de 2004 sur les circuits *single electron counting* qui utilise cette technique
  - toujours pas besoin de savoir compter

Néolithique

Invention “par les bergers du néolithique”

- du système de numération unaire
  - un caillou, un mouton
  - pas besoin de savoir compter pour manipuler des nombres
- de l’addition
  - algorithme : verser un sac de caillou dans l’autre
  - rigolez pas, il y a un papier de 2004 sur les circuits *single electron counting* qui utilise cette technique
  - toujours pas besoin de savoir compter
- de la soustraction, de la comparaison, ...

Néolithique

Invention “par les bergers du néolithique”

- du système de numération unaire
  - un caillou, un mouton
  - pas besoin de savoir compter pour manipuler des nombres
- de l’addition
  - algorithme : verser un sac de caillou dans l’autre
  - rigolez pas, il y a un papier de 2004 sur les circuits *single electron counting* qui utilise cette technique
  - toujours pas besoin de savoir compter
- de la soustraction, de la comparaison, ...

En latin, *calculus* signifie petit caillou.

Antiquité

- Le unaire montre ses limites pour représenter de grands nombres.

Antiquité

- Le unaire montre ses limites pour représenter de grands nombres.
- Avec l'invention des inégalités sociales, besoin de systèmes de numération plus évolués.

Antiquité

- Le unaire montre ses limites pour représenter de grands nombres.
- Avec l'invention des inégalités sociales, besoin de systèmes de numération plus évolués.
- Chez nous en Grèce (mais aussi en Égypte et en Chine), développement de systèmes alphabétiques
  - On remplace les cailloux par des pièces de monnaie.

Antiquité

- Le unaire montre ses limites pour représenter de grands nombres.
- Avec l'invention des inégalités sociales, besoin de systèmes de numération plus évolués.
- Chez nous en Grèce (mais aussi en Égypte et en Chine), développement de systèmes alphabétiques
  - On remplace les cailloux par des pièces de monnaie.
  - Chaque symbole a une valeur numérique fixe : I, V, X, C...

Antiquité

- Le unaire montre ses limites pour représenter de grands nombres.
- Avec l'invention des inégalités sociales, besoin de systèmes de numération plus évolués.
- Chez nous en Grèce (mais aussi en Égypte et en Chine), développement de systèmes alphabétiques
  - On remplace les cailloux par des pièces de monnaie.
  - Chaque symbole a une valeur numérique fixe : I, V, X, C...
  - On utilise le unaire pour chaque symbole : XVII

Antiquité

- Le unaire montre ses limites pour représenter de grands nombres.
- Avec l'invention des inégalités sociales, besoin de systèmes de numération plus évolués.
- Chez nous en Grèce (mais aussi en Égypte et en Chine), développement de systèmes alphabétiques
  - On remplace les cailloux par des pièces de monnaie.
  - Chaque symbole a une valeur numérique fixe : I, V, X, C...
  - On utilise le unaire pour chaque symbole : XVII
  - Addition par mélange

Antiquité

- Le unaire montre ses limites pour représenter de grands nombres.
- Avec l'invention des inégalités sociales, besoin de systèmes de numération plus évolués.
- Chez nous en Grèce (mais aussi en Égypte et en Chine), développement de systèmes alphabétiques
  - On remplace les cailloux par des pièces de monnaie.
  - Chaque symbole a une valeur numérique fixe : I, V, X, C...
  - On utilise le unaire pour chaque symbole : XVII
  - Addition par mélange
  - Règles optionnelles pour une représentation canonique

Lisez *Les langages numériques*, de Jean Villemin

D'après un haut relief de Saqqara en Egypte. Les oiseaux indiquent l'ordre de lecture, (..) le bec au vent. Les autres hiéroglyphes comptent (...). Chaque signe vaut une puissance de dix : un pour la barre, 10 pour le fer à cheval, 100 pour le serpent, 1 000 pour le lotus, 10 000 pour l'obélisque et 100 000 pour la salamandre. Les quatre nombres qui figurent ici s'écrivent donc, en décimal, 11 110 (haut gauche), (...).

Antiquité toujours, mais chez les sauvages

Les systèmes alphabétiques montrent leur limites pour

- représenter des nombres de taille arbitraire
- et surtout, calculer dessus.

Antiquité toujours, mais chez les sauvages

Les systèmes alphabétiques montrent leur limites pour

- représenter des nombres de taille arbitraire
- et surtout, calculer dessus.

Invention des systèmes à position (Babylone, Inde, Mayas)

- c'est la position du chiffre dans le nombre qui donne sa puissance de la base
- simplification des algorithmes de calcul

Antiquité toujours, mais chez les sauvages

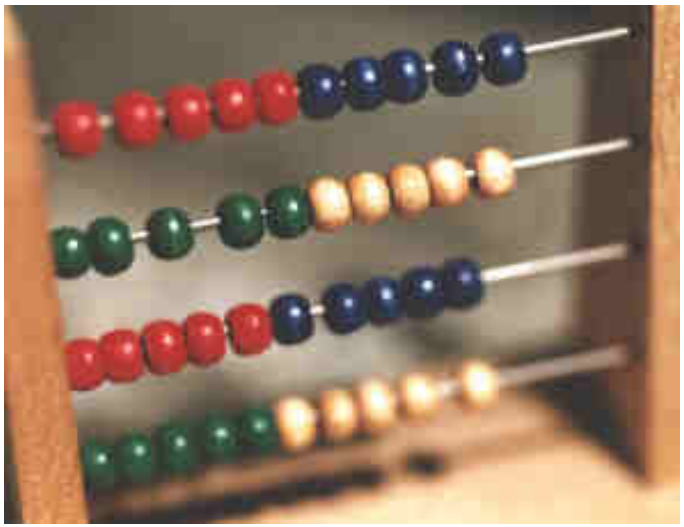
Les systèmes alphabétiques montrent leur limites pour

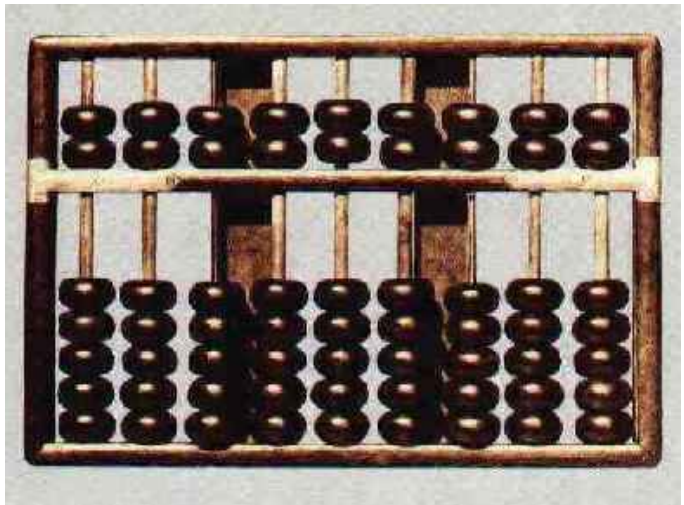
- représenter des nombres de taille arbitraire
- et surtout, calculer dessus.

Invention des systèmes à position (Babylone, Inde, Mayas)

- c'est la position du chiffre dans le nombre qui donne sa puissance de la base
- simplification des algorithmes de calcul
- permet l'invention de la calculette de poche (env -1000)

Une calculette de poche 4 opérations

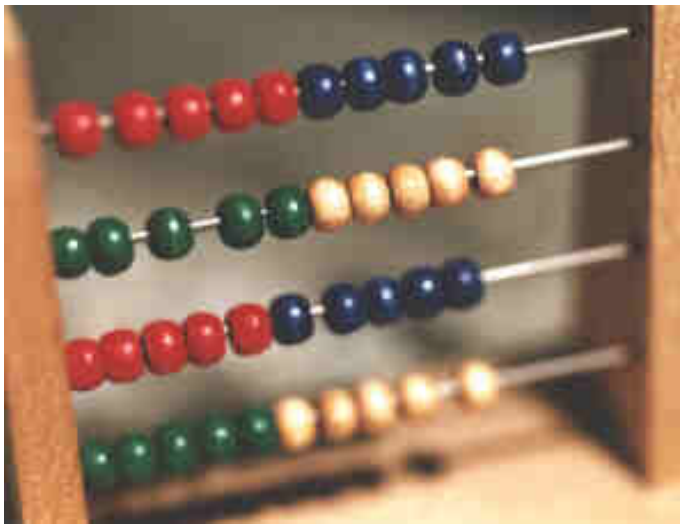


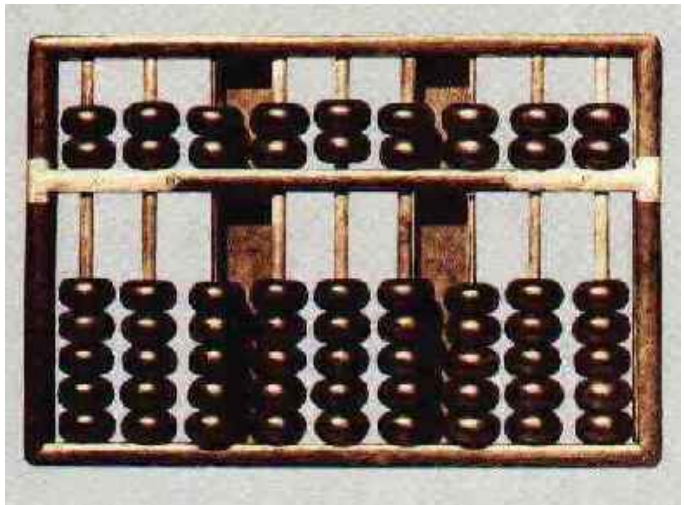


Remarque :

- numération de position en base 10
- mais chaque chiffre est codé en unaire (à gauche) ou en alphabétique (à droite)

Une calculette de poche 4 opérations





Remarque :

- numération de position en base 10
- mais chaque chiffre est codé en unaire (à gauche) ou en alphabétique (à droite)

Mode d'emploi ? C'est un algorithme.

Numération simple de position

Formalisons d'abord le système de représentation des nombres :

- On appelle base un entier β supérieur ou égal à 2.

Numération simple de position

Formalisons d'abord le système de représentation des nombres :

- On appelle base un entier β supérieur ou égal à 2.
- Un chiffre est un entier compris entre 0 et $\beta - 1$.

Numération simple de position

Formalisons d'abord le système de représentation des nombres :

- On appelle base un entier β supérieur ou égal à 2.
- Un chiffre est un entier compris entre 0 et $\beta - 1$.
- La séquence de chiffres $x_{n-1}...x_0$ représente l'entier

$$X = \sum_{i=0}^{n-1} \beta^i x_i$$

Numération simple de position

Formalisons d'abord le système de représentation des nombres :

- On appelle base un entier β supérieur ou égal à 2.
- Un chiffre est un entier compris entre 0 et $\beta - 1$.
- La séquence de chiffres $x_{n-1}...x_0$ représente l'entier

$$X = \sum_{i=0}^{n-1} \beta^i x_i$$

- On appelle poids les puissances de la base.
  Exemple : "Aux zéros de poids fort près, cette représentation est unique"

Numération simple de position

Formalisons d'abord le système de représentation des nombres :

- On appelle base un entier β supérieur ou égal à 2.
- Un chiffre est un entier compris entre 0 et $\beta - 1$.
- La séquence de chiffres $x_{n-1}...x_0$ représente l'entier

$$X = \sum_{i=0}^{n-1} \beta^i x_i$$

- On appelle poids les puissances de la base.
  Exemple : "Aux zéros de poids fort près, cette représentation est unique"

Reste plus qu'à formaliser l'algorithme d'addition.
C'est celui que vous avez appris à l'école.

Le paléo-algorithme d'addition

Soit la fonction table d'addition :

$$\begin{array}{rccl} \text{TA} : & \{0,1\} \times \{0..\beta-1\}^2 & \to & \{0..1\} \times \{0..\beta-1\} \\ & (r,x,y) & \mapsto & (r',s) \quad \text{tq} \quad \beta r' + s = r + x + y \end{array}$$

En francais,

- cette fonction prend une retenue entrante (0 ou 1) et deux chiffres, et retourne leur somme ;

Le paléo-algorithme d'addition

Soit la fonction table d'addition :

$$\begin{array}{rccl} \text{TA} : & \{0,1\} \times \{0..\beta-1\}^2 & \to & \{0..1\} \times \{0..\beta-1\} \\ & (r,x,y) & \mapsto & (r',s) \quad \text{tq} \quad \beta r' + s = r + x + y \end{array}$$

En francais,

- cette fonction prend une retenue entrante (0 ou 1) et deux chiffres, et retourne leur somme ;
- Cette somme est comprise entre 0 et $2\beta - 1$.
  Elle s'écrit donc en base β sur deux chiffres.
- Le chiffre de poids fort de la somme est appelé retenue
  et vaut 0 ou 1 quelle que soit la base.

Le paléo-algorithme d'addition

Soit la fonction table d'addition :

$$\begin{array}{rccl} \text{TA}: & \{0,1\} \times \{0..\beta-1\}^2 & \to & \{0..1\} \times \{0..\beta-1\} \\ & (r,x,y) & \mapsto & (r',s) \quad \text{tq} \quad \beta r' + s = r + x + y \end{array}$$

En francais,

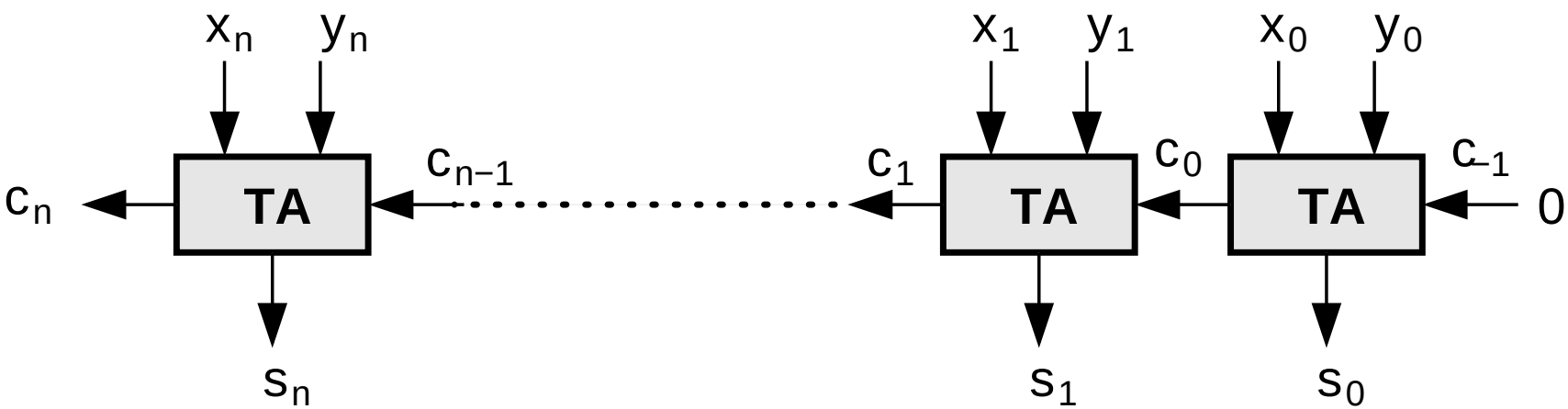
- cette fonction prend une retenue entrante (0 ou 1) et deux chiffres, et retourne leur somme ;
- Cette somme est comprise entre 0 et $2\beta - 1$.
  Elle s'écrit donc en base β sur deux chiffres.
- Le chiffre de poids fort de la somme est appelé retenue
  et vaut 0 ou 1 quelle que soit la base.

L'algorithme d'addition de deux nombres de n chiffres est alors

1: $r_{-1} = 0$
2: **for** $i = 0$ to n **do**
3: $\quad (r_i, s_i) = \text{TA}(r_{i-1}, x_i, y_i)$
4: **end for**

Des opérateurs d'addition

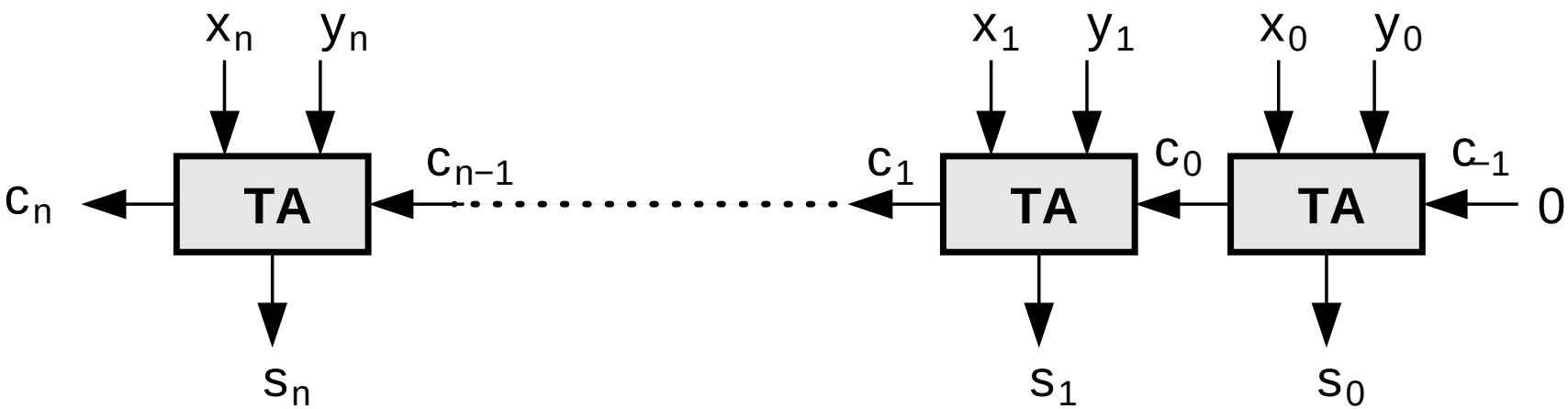
Quelle que soit la techno, il suffit de dérouler l'algorithme précédente pour obtenir un additionneur :



Les flèches indiquent une transmission d'information (dépendance de donnée)

Des opérateurs d'addition

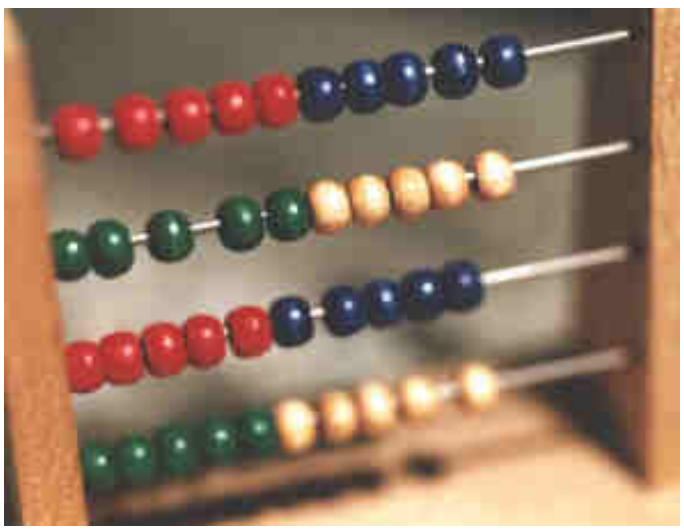
Quelle que soit la techno, il suffit de dérouler l'algorithme précédente pour obtenir un additionneur :

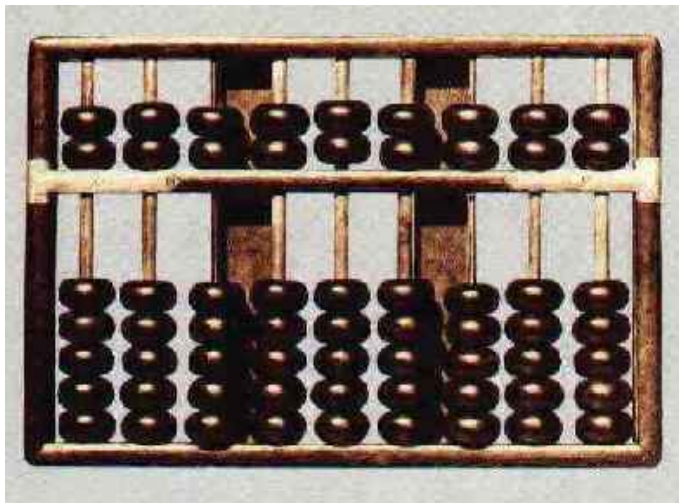


Les flèches indiquent une transmission d'information (dépendance de donnée) : l'algorithme est intrinsèquement séquentiel, et le temps de calcul sera au mieux linéaire en le nombre de chiffres.

Mode d'emploi du boulier

- base $\beta = 10$
- fonction TA : addition en unaire des chiffres
  - mélange des cailloux
- modulo la base et propagation de retenue effectués par les doigts de l'opérateur





State of the art : les nombres à Babylone

- Base 60 parce que c'est divisible par tout (alors que 10, bof)

State of the art : les nombres à Babylone

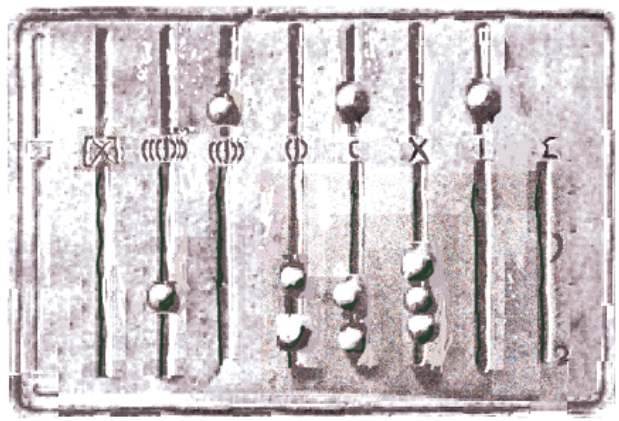
- Base 60 parce que c'est divisible par tout (alors que 10, bof)
- mais chaque chiffre codé en alphabetique à 2 symboles (pour 1 et 10), plus l'espace

State of the art : les nombres à Babylone

- Base 60 parce que c'est divisible par tout (alors que 10, bof)
- mais chaque chiffre codé en alphabetique à 2 symboles (pour 1 et 10), plus l'espace
- abaques/bouliers en décimal, deux colonnes par chiffre

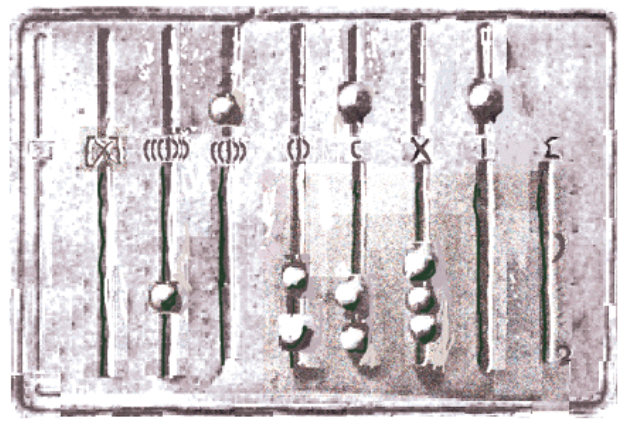
State of the art : les nombres à Babylone

- Base 60 parce que c'est divisible par tout (alors que 10, bof)
- mais chaque chiffre codé en alphabetique à 2 symboles (pour 1 et 10), plus l'espace
- abaques/bouliers en décimal, deux colonnes par chiffre
- Remarque : les Romains aussi avaient des abaques utilisant le système de position en base 10.



State of the art : les nombres à Babylone

- Base 60 parce que c'est divisible par tout (alors que 10, bof)
- mais chaque chiffre codé en alphabetique à 2 symboles (pour 1 et 10), plus l'espace
- abaques/bouliers en décimal, deux colonnes par chiffre
- Remarque : les Romains aussi avaient des abaques utilisant le système de position en base 10.



Le système utilisé pour le calcul peut être différent du système utilisé pour la représentation.

On en est toujours là

Quelques conclusions :

- Ces inventions sont guidées par la nécessité de manipuler les nombres
  - Essayez de représenter 11 110 en unaire
  - Essayez donc de faire une addition en chiffres romains...

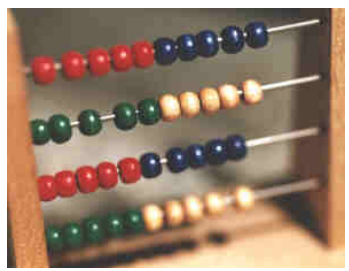
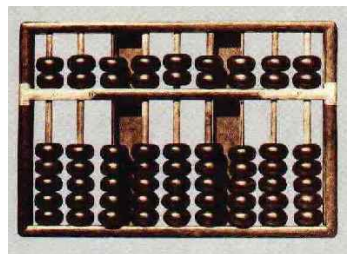
On en est toujours là

Quelques conclusions :

- Ces inventions sont guidées par la nécessité de manipuler les nombres
  - Essayez de représenter 11 110 en unaire
  - Essayez donc de faire une addition en chiffres romains...
- Un bon système est celui qui permet de bien faire les calculs compte tenu de la technologie
  - Pb : on sait discerner d'un coup d'œil des quantités jusqu'à 6

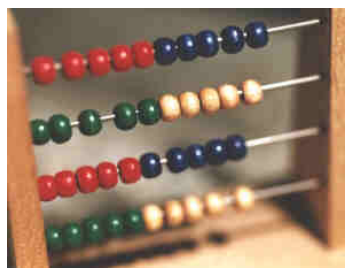
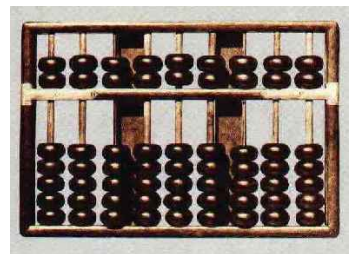
On en est toujours là

Quelques conclusions :

- Ces inventions sont guidées par la nécessité de manipuler les nombres
  - Essayez de représenter 11 110 en unaire
  - Essayez donc de faire une addition en chiffres romains...
- Un bon système est celui qui permet de bien faire les calculs compte tenu de la technologie
  - Pb : on sait discerner d'un coup d'œil des quantités jusqu'à 6
  - Solution :  

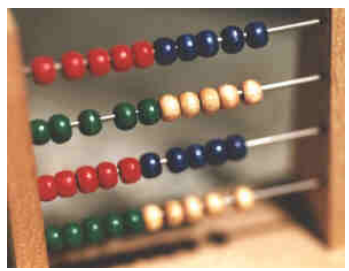
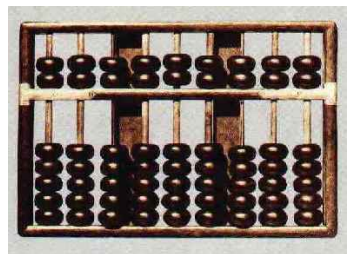
On en est toujours là

Quelques conclusions :

- Ces inventions sont guidées par la nécessité de manipuler les nombres
  - Essayez de représenter 11 110 en unaire
  - Essayez donc de faire une addition en chiffres romains...
- Un bon système est celui qui permet de bien faire les calculs compte tenu de la technologie
  - Pb : on sait discerner d'un coup d'œil des quantités jusqu'à 6
  - Solution :  
- Le système de calcul peut être différent du système de représentation

On en est toujours là

Quelques conclusions :

- Ces inventions sont guidées par la nécessité de manipuler les nombres
  - Essayez de représenter 11 110 en unaire
  - Essayez donc de faire une addition en chiffres romains...
- Un bon système est celui qui permet de bien faire les calculs compte tenu de la technologie
  - Pb : on sait discerner d'un coup d'œil des quantités jusqu'à 6
  - Solution :  
- Le système de calcul peut être différent du système de représentation
- On peut avoir plusieurs couches de représentation

Arithmétique des ordinateurs ?

Et donc l'arithmétique des ordinateurs c'est

- l'études des systèmes de représentation des nombres
- l'études des algorithmes de calcul
- l'étude des implémentations de ces algorithmes

Suite de l'historique

1623 Wilhelm Schickhard invente la propagation de la retenue par roue dentée

- fonction TA (y compris le calcul du modulo) : addition unaire sur les disques
- propagation de retenue automatique : une dent entraîne la roue suivante

Suite de l'historique

1623 Wilhelm Schickhard invente la propagation de la retenue par roue dentée

- fonction TA (y compris le calcul du modulo) : addition unaire sur les disques
- propagation de retenue automatique : une dent entraîne la roue suivante

1624 Il construit la première calculette mécanique à 4 opérations

Suite de l'historique

1623 Wilhelm Schickhard invente la propagation de la retenue par roue dentée

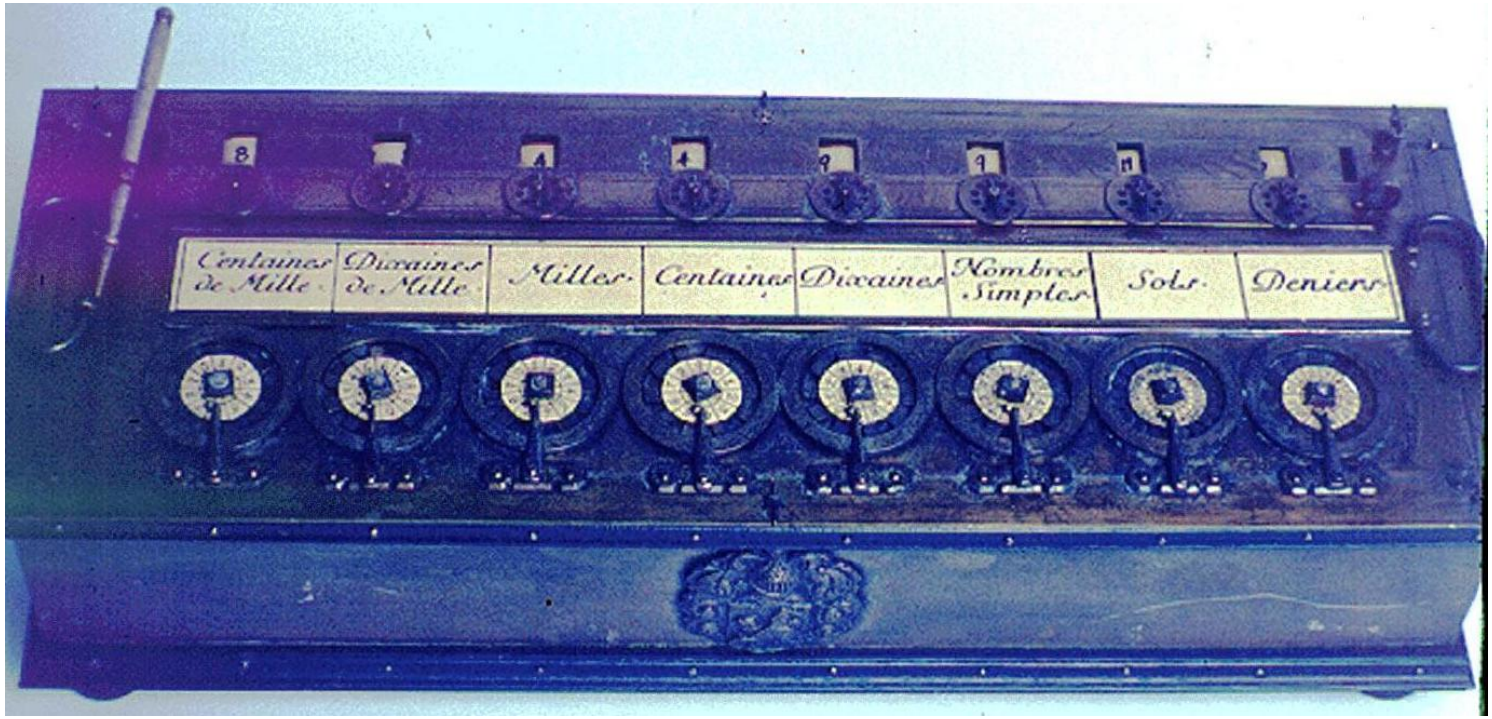
- fonction TA (y compris le calcul du modulo) : addition unaire sur les disques
- propagation de retenue automatique : une dent entraîne la roue suivante

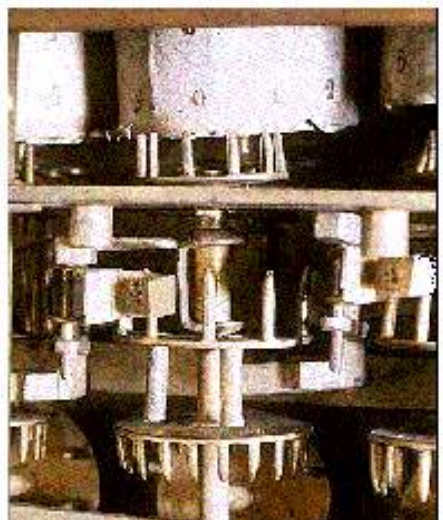
1624 Il construit la première calculette mécanique à 4 opérations

1645 Blaise Pascal aussi

- chez Pascal, un système de poids ou de ressorts régénère le signal lors d'une propagation de retenue longue (9999+1).

La Pascaline

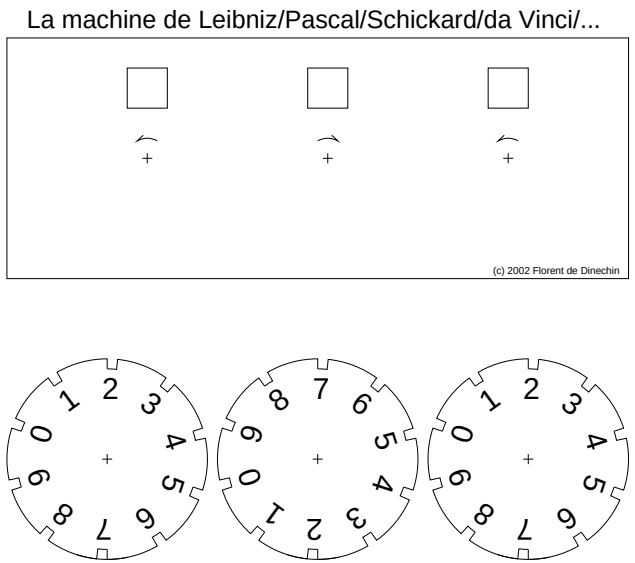




La Pascaline en technologie moderne

... se trouve sur

`http://perso.ens-lyon.fr/florent.de.dinechin/enseignement/pascaline/`



Suite de l'historique

1623 Sir Francis Bacon décrit le codage des nombres dans le système binaire

1679 Gottfried Wilhelm Leibniz écrit *De Progressione Dyadica*, sur l'arithmétique binaire

1822 Charles Babbage se lance dans sa *difference engine*, qui sert à calculer des polynômes

1822 Il laisse tomber car il a une meilleur idée...

1854 George Boole publie sur la logique mathématique

1880 Premières machines à calculer clavier + imprimante

Histoire des calculateurs

1936 Konrad Zuse construit le premier calculateur binaire (à relais)

1937 John V. Atanasoff a l'idée d'utiliser le binaire pour des calculateurs.

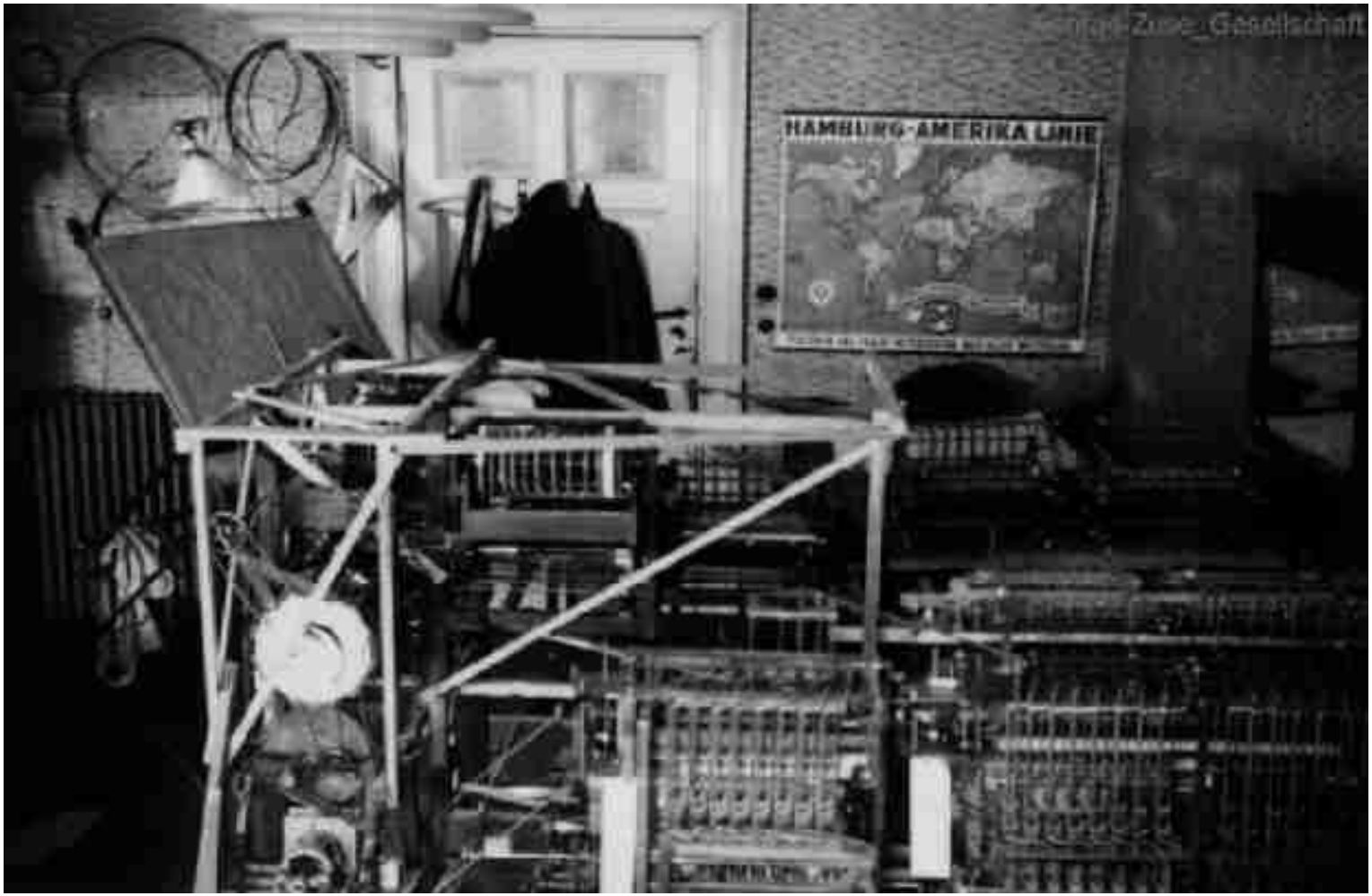
1941 Le Z3 de Zuze est le premier calculateur universel programmable complet

- 1400 relais
- 64 mots de mémoire
- arithmétique binaire virgule flottante sur 22 bits
- programmation par ruban perforé de 8 bits, une boucle
- addition en 50ms, multiplication et division en 3s

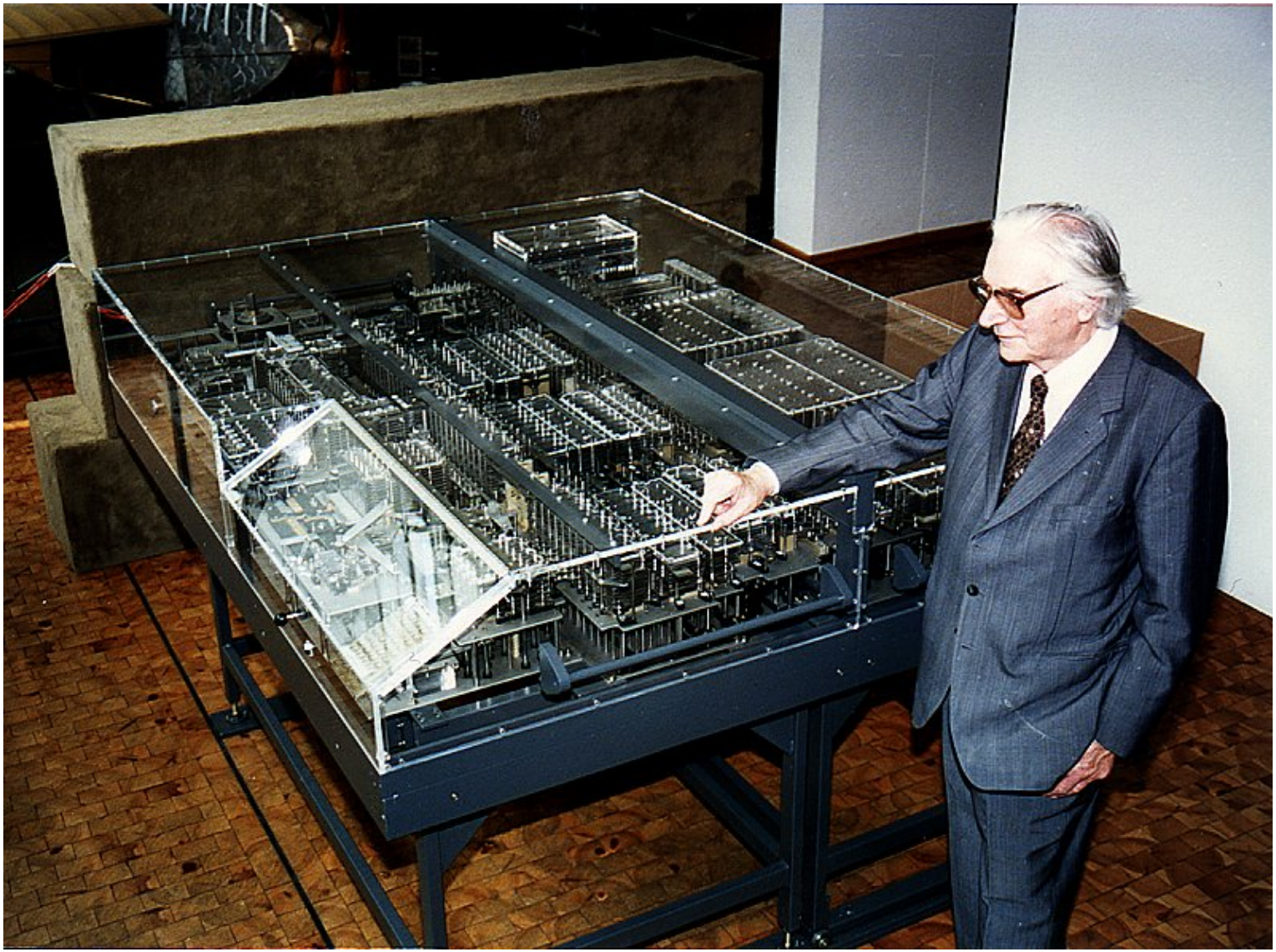
1946 L'ENIAC est le premier calculateur *électronique*, mais sinon c'est un boulier.

1985 Norme IEEE-754 pour la virgule flottante

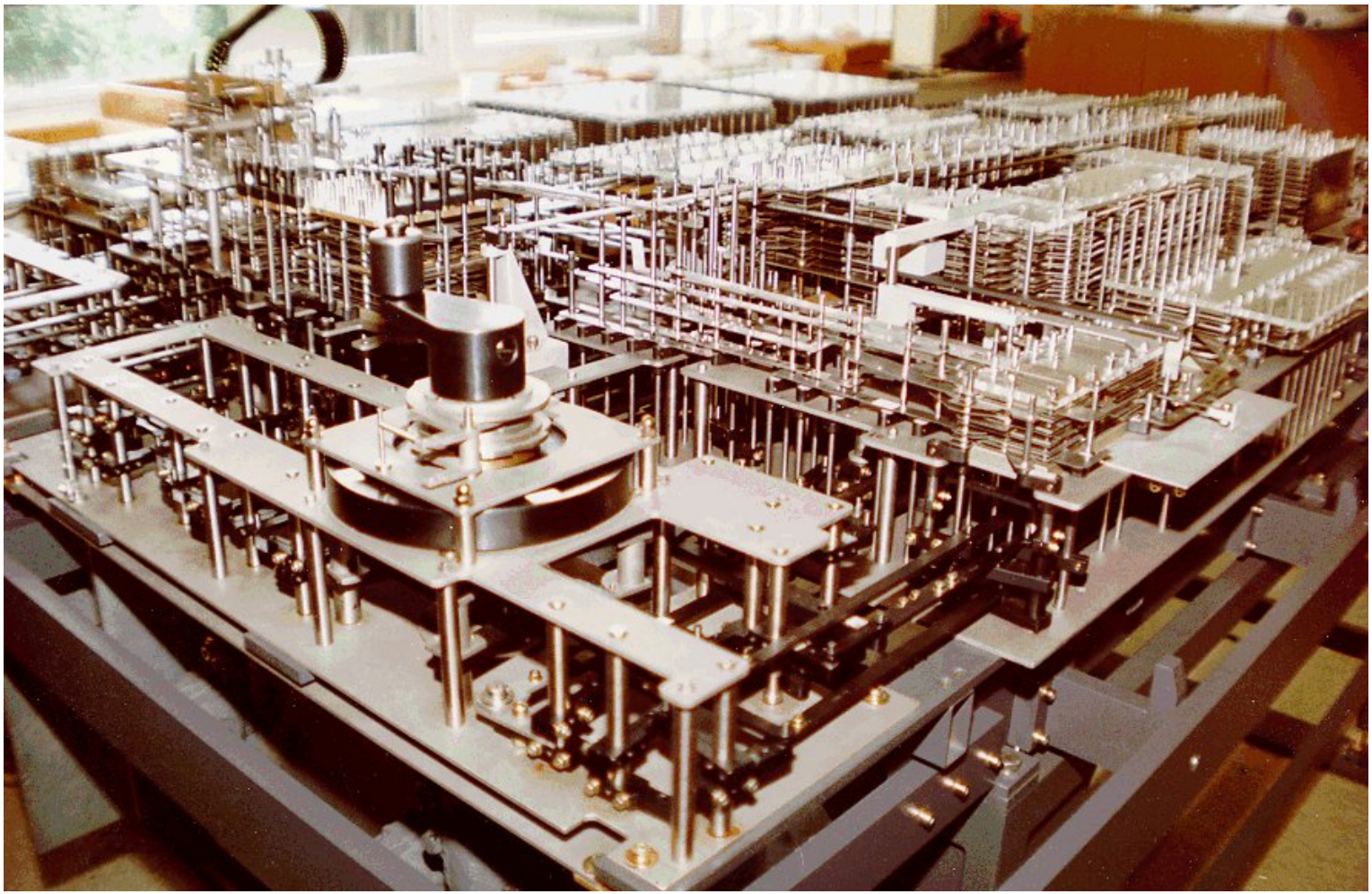
Les calculateurs Zuse



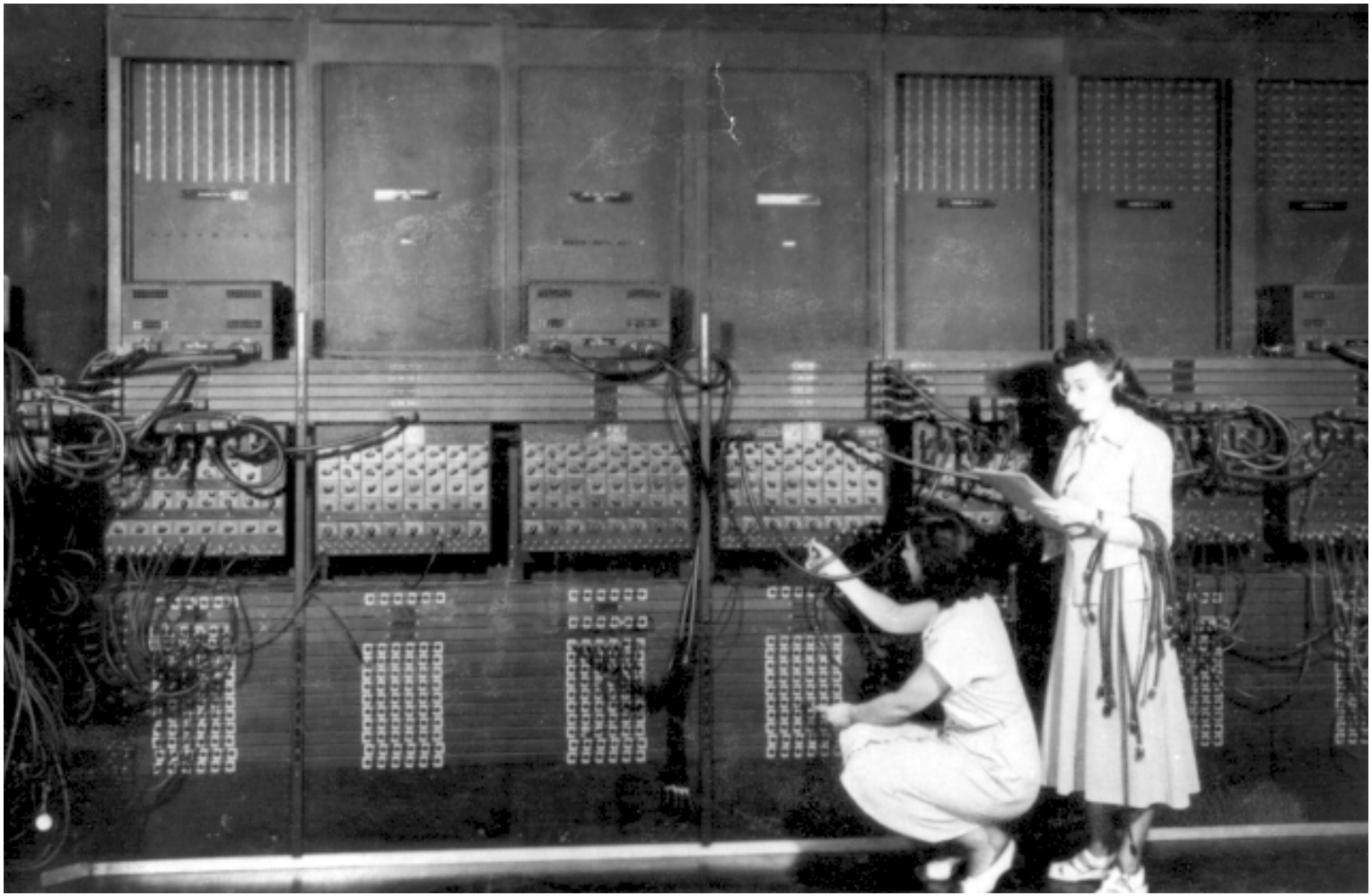
Les calculateurs Zuse



Les calculateurs Zuse



L'Eniac n'avait pas de clavier



Représentations modernes des entiers

Des petits cailloux aux ordinateurs : introduction historique

Représentations modernes des entiers

Représentations des réels

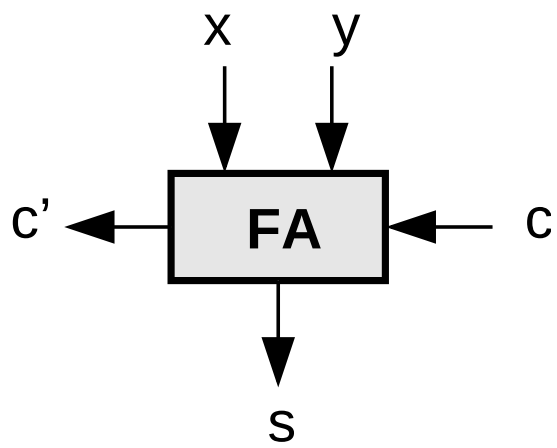
Conclusion : l'arithmétique de mon PC

Rappel

1: $c_{-1} = 0$
2: **for** $i = 0$ to n **do**
3: $\quad (c_i, s_i) = \mathrm{TA}(c_{i-1}, x_i, y_i)$
4: **end for**

Additionneur binaire

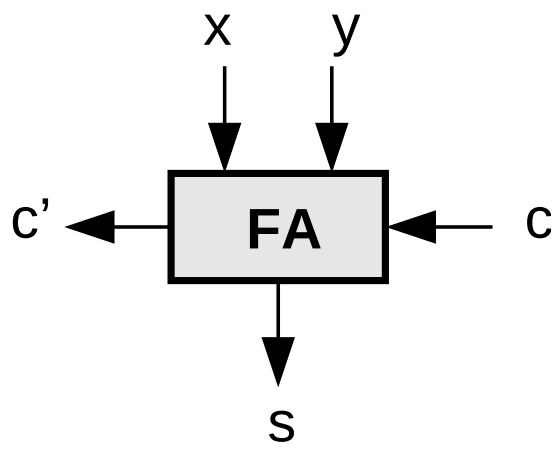
La fonction TA est implémentée par la cellule Full Adder :



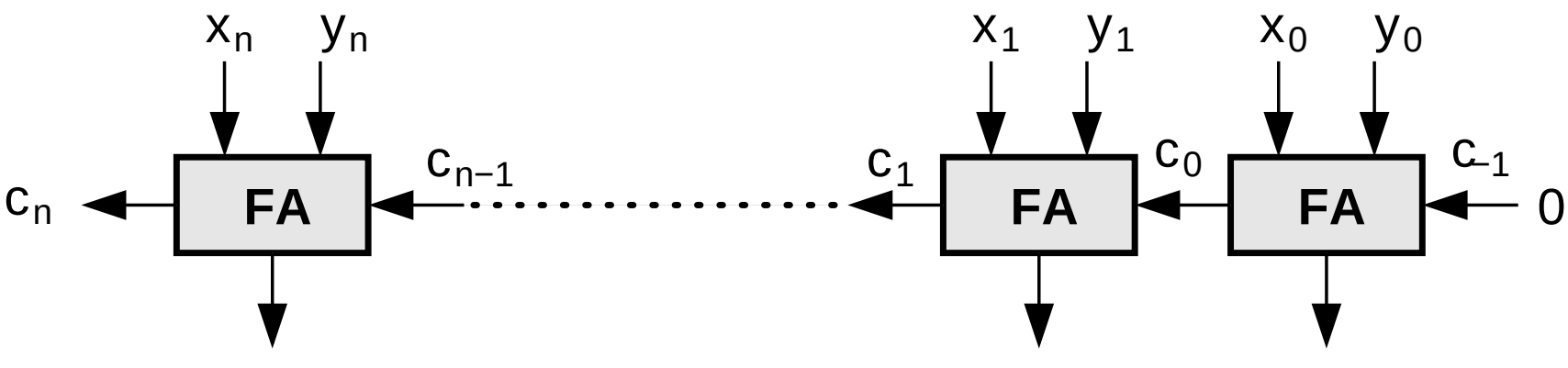
Remarque : cette cellule est fonctionnellement symétrique en ses trois entrées.

Additionneur binaire

La fonction TA est implémentée par la cellule Full Adder :



Remarque : cette cellule est fonctionnellement symétrique en ses trois entrées.
Et l'additionneur recycle le dessin précédent :



La calculette électronique de poche

De même que le boulier fonctionne en *décimal codé en unaire*, la calculette fonctionne le plus souvent en *décimal codé en binaire* (BCD) :

- base 10
- chaque chiffre est lui-même codé en base 2 sur 4 bits
- la fonction TA est construite autour d'un additionneur binaire 4 bits

Ajoutons des nombres de 1000 chiffres

Ce n'est plus du matériel...

- l'addition GMP *(GNU Multiple Precision)* utilise toujours le même algo
- un chiffre est un entier machine (de 16, 32 ou 64 bits)
- autrement dit, calcul en base 2^{16}, 2^{32} ou 2^{64}
- la fonction TA utilise les opérateurs matériels du processeur

C'était un bon exemple

- Arithmétique des ordinateurs :
  - représentation des nombres,
  - algorithmes,
  - implémentations

C'était un bon exemple

- Arithmétique des ordinateurs :
  - représentation des nombres,
  - algorithmes,
  - implémentations
- On a fixé la représentation et fait varier deux paramètres, β et la représentation du chiffre
- On a obtenu plein d'implémentations

C'était un bon exemple

- Arithmétique des ordinateurs :
  - représentation des nombres,
  - algorithmes,
  - implémentations
- On a fixé la représentation et fait varier deux paramètres, β et la représentation du chiffre
- On a obtenu plein d'implémentations
- Techno électronique $\Longrightarrow$ binaire pour les couches basses
  - en attendant mieux
  - (techno mécanique $\Longrightarrow$ unaire pour les couches basses...)

C'était un bon exemple

- Arithmétique des ordinateurs :
  - représentation des nombres,
  - algorithmes,
  - implémentations
- On a fixé la représentation et fait varier deux paramètres, β et la représentation du chiffre
- On a obtenu plein d'implémentations
- Techno électronique $\Longrightarrow$ binaire pour les couches basses
  - en attendant mieux
  - (techno mécanique $\Longrightarrow$ unaire pour les couches basses...)
- Compromis et LCDE
  - grande base $\Longrightarrow$ moins de propagations de retenues, mais TA plus compliquée

C'était un bon exemple

- Arithmétique des ordinateurs :
  - représentation des nombres,
  - algorithmes,
  - implémentations
- On a fixé la représentation et fait varier deux paramètres, β et la représentation du chiffre
- On a obtenu plein d'implémentations
- Techno électronique $\Longrightarrow$ binaire pour les couches basses
  - en attendant mieux
  - (techno mécanique $\Longrightarrow$ unaire pour les couches basses...)
- Compromis et LCDE
  - grande base $\Longrightarrow$ moins de propagations de retenues, mais TA plus compliquée
- Un bon système de représentation permet de bien faire les calculs.

On a fixé la représentation (...)

L'addition dans le système de numération de position est séquentielle, chiffre à chiffre.

- Votre gamin s'en fout, il ne sait faire qu'une chose à la fois.
- L'utilisateur du boulier aussi, il n'a que deux mains.

On a fixé la représentation (...)

L'addition dans le système de numération de position est séquentielle, chiffre à chiffre.

- Votre gamin s'en fout, il ne sait faire qu'une chose à la fois.
- L'utilisateur du boulier aussi, il n'a que deux mains.
- Mais les petits transistors peuvent fonctionner en parallèle.

On a fixé la représentation (...)

L'addition dans le système de numération de position est séquentielle, chiffre à chiffre.

- Votre gamin s'en fout, il ne sait faire qu'une chose à la fois.
- L'utilisateur du boulier aussi, il n'a que deux mains.
- Mais les petits transistors peuvent fonctionner en parallèle.

Il va falloir changer de système de numération.

On a fixé la représentation (...)

L'addition dans le système de numération de position est séquentielle, chiffre à chiffre.

- Votre gamin s'en fout, il ne sait faire qu'une chose à la fois.
- L'utilisateur du boulier aussi, il n'a que deux mains.
- Mais les petits transistors peuvent fonctionner en parallèle.

Il va falloir changer de système de numération.

(augmenter la base permettait déjà d'avoir moins de propagations de retenue)

Une petite escroquerie pour commencer

Voici un nouveau système de représentation des nombres :

- Un nombre entier X peut être représenté par une séquence de couples (c_i, s_i)

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

Une petite escroquerie pour commencer

Voici un nouveau système de représentation des nombres :

- Un nombre entier X peut être représenté par une séquence de couples (c_i, s_i)

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

- Cette représentation n'est plus unique.
  Et alors ? Cela s'appelle un codage redondant

Une petite escroquerie pour commencer

Voici un nouveau système de représentation des nombres :

- Un nombre entier X peut être représenté par une séquence de couples (c_i, s_i)

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

- Cette représentation n'est plus unique.
  Et alors ? Cela s'appelle un codage redondant
- On sait passer de cette représentation à la représentation standard. Comment ?

Une petite escroquerie pour commencer

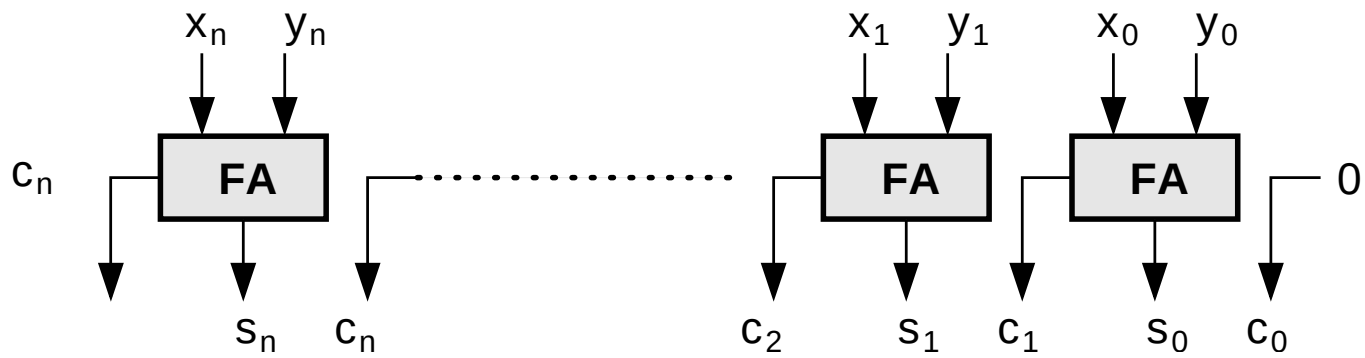
Voici un nouveau système de représentation des nombres :

- Un nombre entier X peut être représenté par une séquence de couples (c_i, s_i)

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

- Cette représentation n'est plus unique.
  Et alors ? Cela s'appelle un codage redondant
- On sait passer de cette représentation à la représentation standard. Comment ?
- Construisons un additionneur qui prend deux nombres en binaire normal, et renvoie sont résultat dans ce système de numération...

Notre premier additionneur parallèle

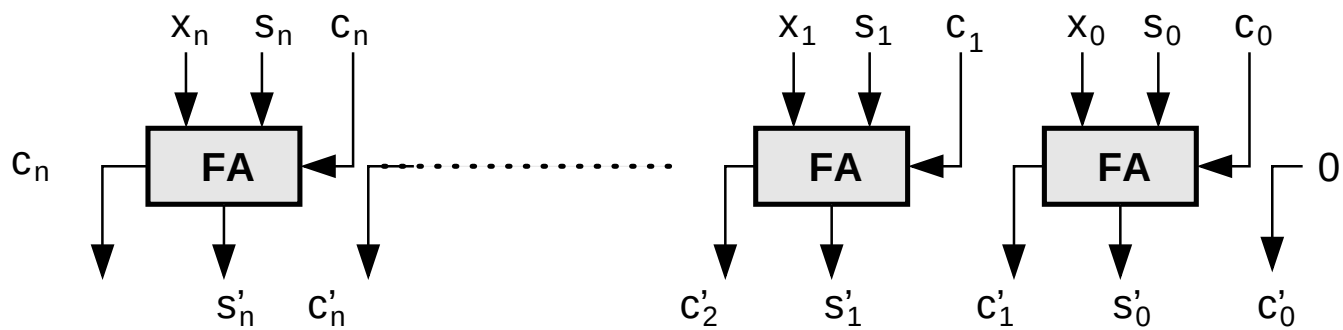


Cette représentation s'appelle *carry-save*, ou représentation à retenue conservée.

Constatez avec ravissement qu'en changeant de représentation, on s'est débarassé de cette fâcheuse propagation de retenue !

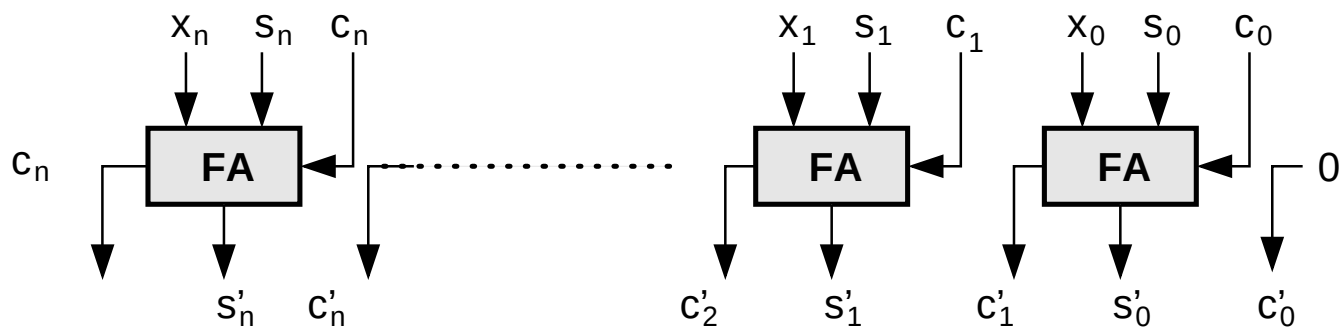
Et pour le même prix

Le même additionneur sait additionner un nombre en *carry-save* et un nombre en binaire normal, et produire son résultat en *carry-save*



Et pour le même prix

Le même additionneur sait additionner un nombre en *carry-save* et un nombre en binaire normal, et produire son résultat en *carry-save*



Et toujours pas de propagation de retenue !

Amoralité

Ce n'était pas une escroquerie :

- Si vous avez plein de nombres en binaires à additionner, conservez votre somme partielle en *carry-save*.
- Toutes vos additions intermédiaires seront parallèles.

Amoralité

Ce n'était pas une escroquerie :

- Si vous avez plein de nombres en binaires à additionner, conservez votre somme partielle en *carry-save*.
- Toutes vos additions intermédiaires seront parallèles.
- (mais bon, le résultat sera en *carry-save*...)

Amoralité

Ce n'était pas une escroquerie :

- Si vous avez plein de nombres en binaires à additionner, conservez votre somme partielle en *carry-save*.
- Toutes vos additions intermédiaires seront parallèles.
- (mais bon, le résultat sera en *carry-save*...)
- Si vous le voulez en binaire normal, il faut faire une addition à propagation de retenue tout de même.

Amoralité

Ce n'était pas une escroquerie :

- Si vous avez plein de nombres en binaires à additionner, conservez votre somme partielle en *carry-save*.
- Toutes vos additions intermédiaires seront parallèles.
- (mais bon, le résultat sera en *carry-save*...)
- Si vous le voulez en binaire normal, il faut faire une addition à propagation de retenue tout de même.

Eh bien c'est utilisé dans des multiplieurs, des filtres...

Au fait, la multiplication

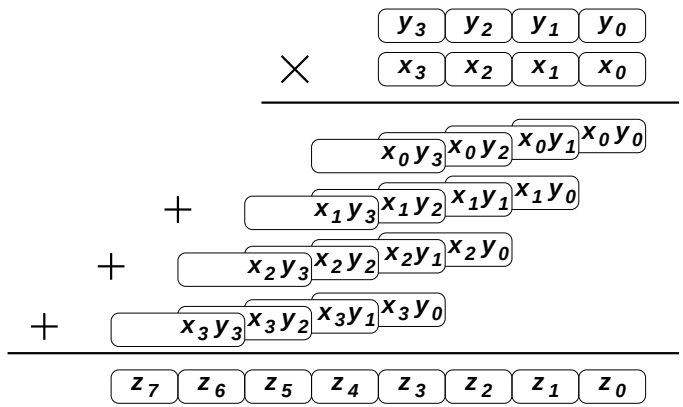
Comme pour l'algo d'addition,

- On part de l'algorithme à la main
- On essaye d'enlever la séquentialité due à la condition humaine

Au fait, la multiplication

Comme pour l'algo d'addition,

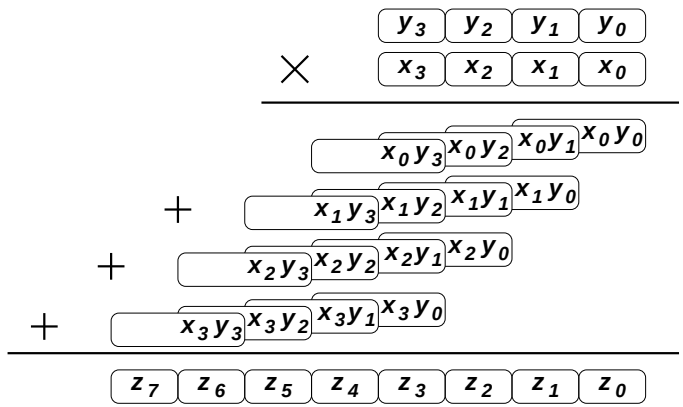
- On part de l'algorithme à la main
- On essaye d'enlever la séquentialité due à la condition humaine
  - Aide : la multiplication est commutative, il faut faire un dessin commutatif



Au fait, la multiplication

Comme pour l'algo d'addition,

- On part de l'algorithme à la main
- On essaye d'enlever la séquentialité due à la condition humaine
  - Aide : la multiplication est commutative, il faut faire un dessin commutatif

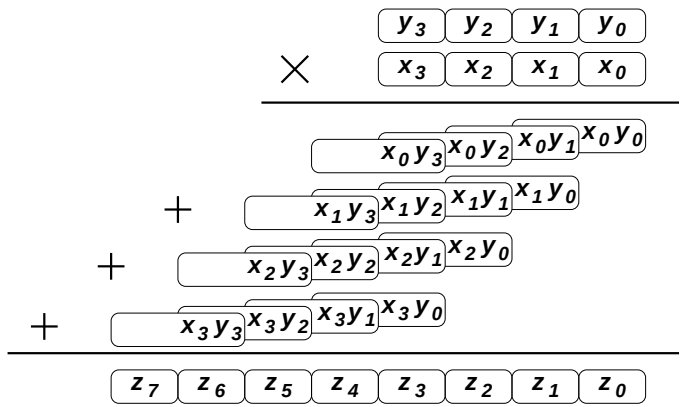


- Le problème se ramène à l'addition de plein de produits partiels

Au fait, la multiplication

Comme pour l'algo d'addition,

- On part de l'algorithme à la main
- On essaye d'enlever la séquentialité due à la condition humaine
  - Aide : la multiplication est commutative, il faut faire un dessin commutatif



- Le problème se ramène à l'addition de plein de produits partiels
- On peut faire varier la base, de 2 à 2^{64}

Éloge de la redondance (1)

C'est la redondance de notre représentation *carry-save* qui lui permet de manger les retenues sans les propager.

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

On peut généraliser.

Éloge de la redondance (1)

C'est la redondance de notre représentation *carry-save* qui lui permet de manger les retenues sans les propager.

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

On peut généraliser.

- On était en base 2

Éloge de la redondance (1)

C'est la redondance de notre représentation *carry-save* qui lui permet de manger les retenues sans les propager.

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

On peut généraliser.

- On était en base 2
- On avait des chiffres entre 0 et 2

Éloge de la redondance (1)

C'est la redondance de notre représentation *carry-save* qui lui permet de manger les retenues sans les propager.

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

On peut généraliser.

- On était en base 2
- On avait des chiffres entre 0 et 2

On peut se placer en base β quelconque et prendre nos chiffres entre ω et $\omega + q$ (Systèmes d'Avizienis)

Éloge de la redondance (1)

C'est la redondance de notre représentation *carry-save* qui lui permet de manger les retenues sans les propager.

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

On peut généraliser.

- On était en base 2
- On avait des chiffres entre 0 et 2

On peut se placer en base β quelconque et prendre nos chiffres entre ω et $\omega + q$ (Systèmes d'Avizienis)
Sous certaines conditions sur β, ω, q... le système sera redondant et on pourra faire des additions complètement parallèles.

Éloge de la redondance (1)

C'est la redondance de notre représentation *carry-save* qui lui permet de manger les retenues sans les propager.

$$X = \sum_{i=0}^{n-1} 2^i (c_i + s_i)$$

On peut généraliser.

- On était en base 2
- On avait des chiffres entre 0 et 2

On peut se placer en base β quelconque et prendre nos chiffres entre ω et $\omega + q$ (Systèmes d'Avizienis)
Sous certaines conditions sur β, ω, q... le système sera redondant et on pourra faire des additions complètement parallèles.
(mais il y a plein de théorèmes compliqués à démontrer alors je ne détaille pas plus...)

Éloge de la redondance (2)

Plus généralement, la redondance donne de la liberté.

Éloge de la redondance (2)

Plus généralement, la redondance donne de la liberté.
Exemple : la division.

- À la main
  - Il faut “intuiter” le prochain chiffre du quotient.
  - Ensuite on vérifie que c'était le bon en effectuant une multiplication et une soustraction.

Éloge de la redondance (2)

Plus généralement, la redondance donne de la liberté.
Exemple : la division.

- À la main
  - Il faut “intuiter” le prochain chiffre du quotient.
  - Ensuite on vérifie que c'était le bon en effectuant une multiplication et une soustraction.
- Le choix correct du prochain chiffre est une fonction de tous les chiffres du diviseur et du reste courant
  $\longrightarrow$ coûteuse à implémenter

Éloge de la redondance (2)

Plus généralement, la redondance donne de la liberté.
Exemple : la division.

- À la main
  - Il faut "intuiter" le prochain chiffre du quotient.
  - Ensuite on vérifie que c'était le bon en effectuant une multiplication et une soustraction.
- Le choix correct du prochain chiffre est une fonction de tous les chiffres du diviseur et du reste courant
  $\longrightarrow$ coûteuse à implémenter
- Le choix d'une arithmétique redondante pour les calculs intermédiaires permet de se tromper un peu, et de rattraper à l'itération suivante.
  $\longrightarrow$ une fonction de moins de chiffres fait l'affaire

Encore plus parallèle : codage modulaire des entiers

- Soit $\mu_1, \mu_2, ...\mu_n$ des entiers premiers entre eux deux à deux.

- Tout nombre entier X compris entre 0 et $\prod_i \mu_i - 1$
  peut se représenter de manière unique
  par le vecteur $(m_1, m_2...m_n)$ où $m_i \equiv X[\mu_i]$

Encore plus parallèle : codage modulaire des entiers

- Soit $\mu_1, \mu_2, ...\mu_n$ des entiers premiers entre eux deux à deux.
- Tout nombre entier X compris entre 0 et $\prod_i \mu_i - 1$ peut se représenter de manière unique par le vecteur $(m_1, m_2...m_n)$ où $m_i \equiv X[\mu_i]$
- L'addition, la soustraction, la multiplication se font en parallèle sur les m_i.

Exemple

$$(m_1, m_2, m_3) = (2, 3, 5)$$

```
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3
          |                                                             |
```

$$(m_1, m_2, m_3) = (2, 3, 4)$$

```
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
                  |                           |
```

Problème : $(1, 1, 2) < (1, 0, 0)$?

LCDE pour le RNS

- On a gagné l'addition et la multiplication parallèles sur les chiffres.
- On a perdu
  - les ordres de grandeurs
  - donc la comparaison
  - et la détection des dépassements de capacité,
  - et la division
  - donc la conversion en un système de représentation usuel

LCDE pour le RNS

- On a gagné l'addition et la multiplication parallèles sur les chiffres.
- On a perdu
  - les ordres de grandeurs
  - donc la comparaison
  - et la détection des dépassements de capacité,
  - et la division
  - donc la conversion en un système de représentation usuel

Eh ben il reste quand même des applications en traitement du signal et cryptographie

LCDE pour le RNS

- On a gagné l'addition et la multiplication parallèles sur les chiffres.
- On a perdu
  - les ordres de grandeurs
  - donc la comparaison
  - et la détection des dépassements de capacité,
  - et la division
  - donc la conversion en un système de représentation usuel

Eh ben il reste quand même des applications en traitement du signal et cryptographie

Recherches en cours :

- opérations difficiles
- choix des moduli

Conclusion jusque là

- Il n'y a pas que la numération de position dans la vie
- On peut utiliser plusieurs représentations au cours d'une opération
- On peut exprimer plus de parallélisme que dans les algorithmes manuels
- De la redondance dans la représentation donne de la liberté
- Il y a des compromis et des LCDE

Représentations des réels

Des petits cailloux aux ordinateurs : introduction historique

Représentations modernes des entiers

Représentations des réels

Conclusion : l'arithmétique de mon PC

Codage des réels en virgule fixe

- $X = 000123.4567$
- (dans n'importe quelle base)
- Adaptation triviale des opérateurs entiers.

Codages en virgule flottante

$X = 1.234567890.10^2$ (notation scientifique)

- Plus généralement $x = s.m.\beta^e$

Codages en virgule flottante

$X = 1.234567890.10^2$ (notation scientifique)

- Plus généralement $x = s.m.\beta^e$
- Codages de m (mantisse) et e (exposant) :
  - voir les épisodes précédents.

Codages en virgule flottante

$X = 1.234567890.10^2$ (notation scientifique)

- Plus généralement $x = s.m.\beta^e$
- Codages de m (mantisse) et e (exposant) :
  - voir les épisodes précédents.
- $+0$ et -0, sont-ils égaux ou non ?

Codages en virgule flottante

$X = 1.234567890.10^2$ (notation scientifique)

- Plus généralement $x = s.m.\beta^e$
- Codages de m (mantisse) et e (exposant) :
  - voir les épisodes précédents.
- $+0$ et -0, sont-ils égaux ou non ?
- $0.00123.10^3 = 1.23000.10^0$.
  - Pour utiliser toute la précision disponible, on favorise l'écriture où m ne commence pas par un zéro ("nombre normalisés")

Codages en virgule flottante

$X = 1.234567890.10^2$ (notation scientifique)

- Plus généralement $x = s.m.\beta^e$
- Codages de m (mantisse) et e (exposant) :
  - voir les épisodes précédents.
- $+0$ et -0, sont-ils égaux ou non ?
- $0.00123.10^3 = 1.23000.10^0$.
  - Pour utiliser toute la précision disponible, on favorise l'écriture où m ne commence pas par un zéro ("nombre normalisés")
- Norme IEEE-754 (binaire).

Codages en virgule flottante

$X = 1.234567890.10^2$ (notation scientifique)

- Plus généralement $x = s.m.\beta^e$
- Codages de m (mantisse) et e (exposant) :
  - voir les épisodes précédents.
- $+0$ et -0, sont-ils égaux ou non ?
- $0.00123.10^3 = 1.23000.10^0$.
  - Pour utiliser toute la précision disponible, on favorise l'écriture où m ne commence pas par un zéro ("nombre normalisés")
- Norme IEEE-754 (binaire).
- Sa révision en cours standardise aussi la base 10.

Opérations en virgule flottante

- Addition, soustraction, multiplication et division se ramènent à des opérations sur les mantisses et les exposants
- Opérations pas toujours exactes : le résultat doit être arrondi
- On sait réaliser avec un coût raisonnable “le meilleur arrondi possible” pour les quatre opérations.
- On ne sait pas toujours le faire pour les fonctions élémentaires (Dilemme du Fabricant de Tables)

La virgule flottante est une science

- L'ensemble des flottants, ce n'est pas $(\mathbb{R}, +, \times)$

La virgule flottante est une science

- L'ensemble des flottants, ce n'est pas $(\mathbb{R}, +, \times)$
- Un nombre flottant n'est pas une vague approximation d'un classe floue de réels, mais un rationnel bien défini.

La virgule flottante est une science

- L'ensemble des flottants, ce n'est pas $(\mathbb{R}, +, \times)$
- Un nombre flottant n'est pas une vague approximation d'un classe floue de réels, mais un rationnel bien défini.
- Certes on fait des erreurs d'arrondi, mais on sait les caractériser exactement

La virgule flottante est une science

- L'ensemble des flottants, ce n'est pas $(\mathbb{R}, +, \times)$
- Un nombre flottant n'est pas une vague approximation d'un classe floue de réels, mais un rationnel bien défini.
- Certes on fait des erreurs d'arrondi, mais on sait les caractériser exactement
- ... et on sait aussi les compenser

La virgule flottante est une science

- L'ensemble des flottants, ce n'est pas $(\mathbb{R}, +, \times)$
- Un nombre flottant n'est pas une vague approximation d'un classe floue de réels, mais un rationnel bien défini.
- Certes on fait des erreurs d'arrondi, mais on sait les caractériser exactement
- ... et on sait aussi les compenser
- On sait aussi prouver que certaines opérations sont exactes

La virgule flottante est une science

- L'ensemble des flottants, ce n'est pas $(\mathbb{R}, +, \times)$
- Un nombre flottant n'est pas une vague approximation d'un classe floue de réels, mais un rationnel bien défini.
- Certes on fait des erreurs d'arrondi, mais on sait les caractériser exactement
- ... et on sait aussi les compenser
- On sait aussi prouver que certaines opérations sont exactes
- Ne pas dire "l'erreur sur toto" mais dire "la différence entre toto et tata"...
  et bien définir tata.

La virgule flottante est une science

- L'ensemble des flottants, ce n'est pas $(\mathbb{R}, +, \times)$
- Un nombre flottant n'est pas une vague approximation d'un classe floue de réels, mais un rationnel bien défini.
- Certes on fait des erreurs d'arrondi, mais on sait les caractériser exactement
- ... et on sait aussi les compenser
- On sait aussi prouver que certaines opérations sont exactes
- Ne pas dire "l'erreur sur toto" mais dire "la différence entre toto et tata"...
  et bien définir tata.

Et maintenant, une page de publicité.

Pub

Gappa, un générateur automatique de preuves de programmes arithmétiques.
`http://www.ens-lyon.fr/LIP/Arenaire`

Pub

Gappa, un générateur automatique de preuves de programmes arithmétiques.
`http://www.ens-lyon.fr/LIP/Arenaire`

Un outil qui vous met le nez dedans.

L'ensemble des flottants, ce n'est pas $(\mathbb{R}, +, \times)$

```
var A,B : real ;
begin
    A := 1.0 ; B := 1.0 ;
    while ((A + B) - A) - B = 0.0  do A := 2 * A ;
    while ((A + B) - A) - B ≠ 0.0  do B := B + 1.0 ;
end
```

Montrez que ce programme termine.
Quelle est alors la valeur de B ?

Codages logarithmiques

- $x = s.\beta^{l_x}$ où $l_x = \log_\beta x$

Codages logarithmiques

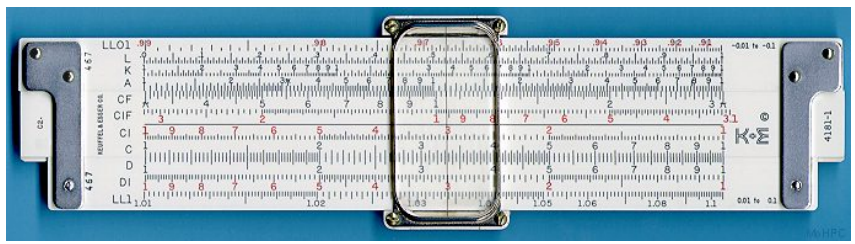
- $x = s.\beta^{l_x}$ où $l_x = \log_\beta x$
- l_x est représenté en virgule fixe signée

Codages logarithmiques

- $x = s.\beta^{l_x}$ où $l_x = \log_\beta x$
- l_x est représenté en virgule fixe signée
- Précision et intervalle représentables similaires au flottant

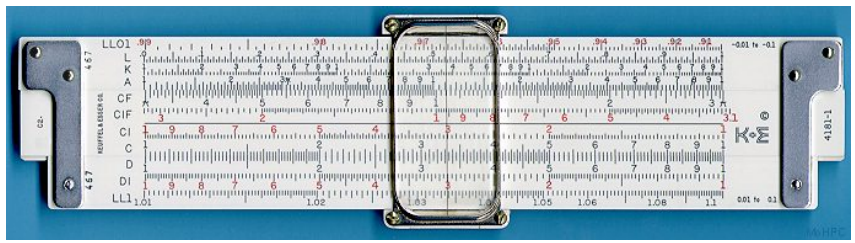
Codages logarithmiques

- $x = s.\beta^{l_x}$ où $l_x = \log_\beta x$
- l_x est représenté en virgule fixe signée
- Précision et intervalle représentables similaires au flottant
- Multiplication, division et racine carrée triviales



Codages logarithmiques

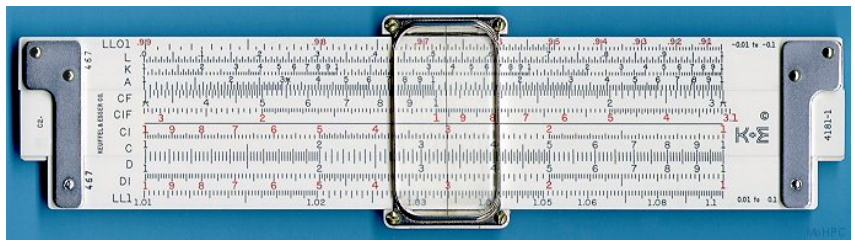
- $x = s.\beta^{l_x}$ où $l_x = \log_\beta x$
- l_x est représenté en virgule fixe signée
- Précision et intervalle représentables similaires au flottant
- Multiplication, division et racine carrée triviales



- LCDE : addition et soustraction difficiles
  - Il faut juste savoir calculer $\log_\beta(1 \pm \beta^r)$

Codages logarithmiques

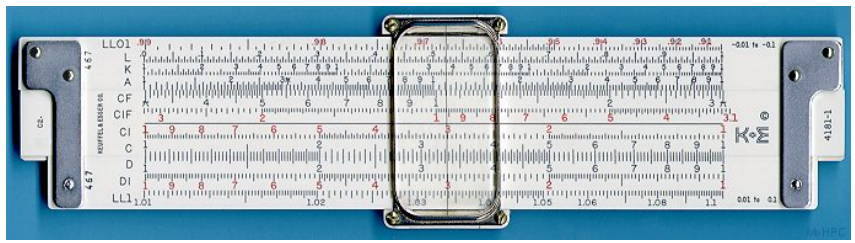
- $x = s.\beta^{l_x}$ où $l_x = \log_\beta x$
- l_x est représenté en virgule fixe signée
- Précision et intervalle représentables similaires au flottant
- Multiplication, division et racine carrée triviales



- LCDE : addition et soustraction difficiles
  - Il faut juste savoir calculer $\log_\beta(1 \pm \beta^r)$
- Multiplication et division exactes

Codages logarithmiques

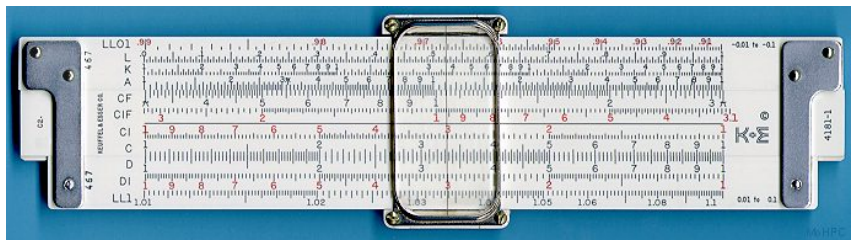
- $x = s.\beta^{l_x}$ où $l_x = \log_\beta x$
- l_x est représenté en virgule fixe signée
- Précision et intervalle représentables similaires au flottant
- Multiplication, division et racine carrée triviales



- LCDE : addition et soustraction difficiles
  - Il faut juste savoir calculer $\log_\beta(1 \pm \beta^r)$
- Multiplication et division exactes
- Addition et soustraction inexactes, et difficiles à spécifier

Codages logarithmiques

- $x = s.\beta^{l_x}$ où $l_x = \log_\beta x$
- l_x est représenté en virgule fixe signée
- Précision et intervalle représentables similaires au flottant
- Multiplication, division et racine carrée triviales



- LCDE : addition et soustraction difficiles
  - Il faut juste savoir calculer $\log_\beta(1 \pm \beta^r)$
- Multiplication et division exactes
- Addition et soustraction inexactes, et difficiles à spécifier

Et maintenant, une page de publicité.

Pub

FPLibrary, une bibliothèque d'opérateurs de base pour la virgule flottante et le système logarithmique en VHDL.

`http://www.ens-lyon.fr/LIP/Arenaire`

Pour une comparaison des deux arithmétiques
au cas par cas des applications.

Arithmétique d'intervalles

- On ne représente plus un réel par un rationnel flottant, mais par deux flottants qui l'encadrent à coup sûr.

Arithmétique d'intervalles

- On ne représente plus un réel par un rationnel flottant, mais par deux flottants qui l'encadrent à coup sûr.
- Les opérations doivent conserver ce "à coup sûr".
  - En langage sérieux : l'addition de deux intervalles $[A_-, A_+]$ et $[B_-, B_+]$ retourne un intervalle $[C_-, C_+]$ qui doit vérifier $\forall a \in [A_-, A_+], \forall b \in [B_-, B_+], \quad a + b \in [C_-, C_+]$

Arithmétique d'intervalles

- On ne représente plus un réel par un rationnel flottant, mais par deux flottants qui l'encadrent à coup sûr.
- Les opérations doivent conserver ce "à coup sûr".
  - En langage sérieux : l'addition de deux intervalles $[A_-, A_+]$ et $[B_-, B_+]$ retourne un intervalle $[C_-, C_+]$ qui doit vérifier $\forall a \in [A_-, A_+], \forall b \in [B_-, B_+], \quad a + b \in [C_-, C_+]$
  - En pratique : $C_- = \bigtriangledown(A_- + B_-),\ C_+ = \bigtriangleup(A_+ + B_+)$

Arithmétique d'intervalles

- On ne représente plus un réel par un rationnel flottant, mais par deux flottants qui l'encadrent à coup sûr.
- Les opérations doivent conserver ce "à coup sûr".
  - En langage sérieux : l'addition de deux intervalles $[A_-, A_+]$ et $[B_-, B_+]$ retourne un intervalle $[C_-, C_+]$ qui doit vérifier $\forall a \in [A_-, A_+], \forall b \in [B_-, B_+], \quad a + b \in [C_-, C_+]$
  - En pratique : $C_- = \nabla(A_- + B_-),\ C_+ = \triangle(A_+ + B_+)$
  - Pour la division évidemment c'est un peu plus compliqué (signes, zéros...)

Arithmétique d'intervalles

- On ne représente plus un réel par un rationnel flottant, mais par deux flottants qui l'encadrent à coup sûr.
- Les opérations doivent conserver ce "à coup sûr".
  - En langage sérieux : l'addition de deux intervalles $[A_-, A_+]$ et $[B_-, B_+]$ retourne un intervalle $[C_-, C_+]$ qui doit vérifier $\forall a \in [A_-, A_+], \forall b \in [B_-, B_+], \quad a + b \in [C_-, C_+]$
  - En pratique : $C_- = \nabla(A_- + B_-),\ C_+ = \triangle(A_+ + B_+)$
  - Pour la division évidemment c'est un peu plus compliqué (signes, zéros...)
  - ...
- Les opérations sont toujours inexactes, mais cela se voit.

Arithmétique d'intervalles

- On ne représente plus un réel par un rationnel flottant, mais par deux flottants qui l'encadrent à coup sûr.
- Les opérations doivent conserver ce "à coup sûr".
  - En langage sérieux : l'addition de deux intervalles $[A_-, A_+]$ et $[B_-, B_+]$ retourne un intervalle $[C_-, C_+]$ qui doit vérifier $\forall a \in [A_-, A_+], \forall b \in [B_-, B_+], \quad a + b \in [C_-, C_+]$
  - En pratique : $C_- = \nabla(A_- + B_-),\ C_+ = \triangle(A_+ + B_+)$
  - Pour la division évidemment c'est un peu plus compliqué (signes, zéros...)
  - ...
- Les opérations sont toujours inexactes, mais cela se voit.
- On peut transformer (presque automatiquement) un programme flottant en un programme par intervalles.
  - on obtient en général un programme qui calcule que $y \in [-\infty, +\infty]$

Arithmétique d'intervalles

- On ne représente plus un réel par un rationnel flottant, mais par deux flottants qui l'encadrent à coup sûr.
- Les opérations doivent conserver ce "à coup sûr".
  - En langage sérieux : l'addition de deux intervalles $[A_-, A_+]$ et $[B_-, B_+]$ retourne un intervalle $[C_-, C_+]$ qui doit vérifier $\forall a \in [A_-, A_+], \forall b \in [B_-, B_+], \quad a + b \in [C_-, C_+]$
  - En pratique : $C_- = \nabla(A_- + B_-),\ C_+ = \triangle(A_+ + B_+)$
  - Pour la division évidemment c'est un peu plus compliqué (signes, zéros...)
  - ...
- Les opérations sont toujours inexactes, mais cela se voit.
- On peut transformer (presque automatiquement) un programme flottant en un programme par intervalles.
  - on obtient en général un programme qui calcule que $y \in [-\infty, +\infty]$
  - pour obtenir un programme utile il faut travailler.

Conclusion

Manipuler des réels est plus risqué que manipuler des entiers,
mais on a des outils

- majorations d'erreur (*forward* et *backward*)
- arithmétique d'intervalle
- arithmétique stochastique
- ...

Conclusion :

l'arithmétique de mon PC

Des petits cailloux aux ordinateurs : introduction historique

Représentations modernes des entiers

Représentations des réels

Conclusion : l'arithmétique de mon PC

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...

Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2
  - mais en plus, 127+1 = -128

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2 0
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2 0
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...
- `int` (abbréviation de *integer*, un terme mathématique)
  - Un entier signé sur 8, 16, 32 ou 64 bits

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2 0
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...
- `int` (abbréviation de *integer*, un terme mathématique)
  - Un entier signé sur 8, 16, 32 ou 64 bits
  - (donc pas vraiment un entier)

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2 0
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...
- `int` (abbréviation de *integer*, un terme mathématique)
  - Un entier signé sur 8, 16, 32 ou 64 bits
  - (donc pas vraiment un entier)
- `float` (un terme informatique, enfin)
  - un flottant 32 bits

On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 🄯
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...
- `int` (abbréviation de *integer*, un terme mathématique)
  - Un entier signé sur 8, 16, 32 ou 64 bits
  - (donc pas vraiment un entier)
- `float` (un terme informatique, enfin)
  - un flottant 32 bits
- `double` (un terme rien du tout)
  - un flottant 64 bits
  - pour les flottants 80 bits essayez `long double` (ha ha)

Blague à part

Les normes (C99, Fortran 2002, IEEE-754) définissent les choses de plus en plus précisément.

Opérations sur ces types de base

- Entiers (32 ou 64 bits)
  - addition/soustraction en un cycle
  - multiplication en quelques cycles, pipelinée
  - (parfois prise en charge par le pipeline flottant)
  - division/modulo en encore plus de cycles, pas pipelinée
  - c'est tout
- Virgule flottante
  - addition en 3-5 cycles, pipelinés
  - multiplication, idem
  - Parfois, *Fused Multiply and Add*
  - division, racine carrée en qq dizaines de cycles, pas pipelinée
  - fonctions élémentaires en qq dizaines (IPF) ou qq centaines (x86) de cycles, pas pipelinées
- Autres
  - vecteurs de petits entiers ou de petits flottants

Portabilité contre performance

- Plus petit dénominateur commun matériel :
  - entiers 32 bits signés ou non
  - flottants simple et double précision
  - 4 opérations, fussent-elles en logiciel

Portabilité contre performance

- Plus petit dénominateur commun matériel :
  - entiers 32 bits signés ou non
  - flottants simple et double précision
  - 4 opérations, fussent-elles en logiciel
- Les possibilités plus exotiques (FMA, doubles étendus, vecteurs) sont mal supportées par les langages/compilateurs
  - même quand elles sont omniprésentes (*add-with-carry*)
  - même quand elles sont faciles à émuler (entiers 64 bits)

Portabilité contre performance

- Plus petit dénominateur commun matériel :
  - entiers 32 bits signés ou non
  - flottants simple et double précision
  - 4 opérations, fussent-elles en logiciel
- Les possibilités plus exotiques (FMA, doubles étendus, vecteurs) sont mal supportées par les langages/compilateurs
  - même quand elles sont omniprésentes (*add-with-carry*)
  - même quand elles sont faciles à émuler (entiers 64 bits)
- Cela a tendance à s'arranger...

Autres cibles technologiques

- ASIC
- FPGA
- GPU
- et même des nanocircuits

C'est assez pour bien travailler

Des questions ?